Homework 11 Selected Solutions
APPM/MATH 4650 Fall '20 Numerical Analysis

Due date: Saturday, December 5, before midnight, via Gradescope. **Instructor**: Prof. Becker
Theme: Absolute stability, Gaussian elimination

solutions version 12/6/2020

Problem 1: Absolute stability of RK4. Recall that the RK4 method is defined by

$$\begin{aligned} k_1 &= hf(t_i, w_i) \\ k_2 &= hf(t_i + h/2, w_i + k_1/2) \\ k_3 &= hf(t_i + h/2, w_i + k_2/2) \\ k_4 &= hf(t_i + h, w_i + k_3) \\ w_{i+1} &= w_i + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \end{aligned}$$

a) Show that the polynomial $Q(h\lambda)$ (equation 5.66 in Burden and Faires) for RK4 is

$$Q(h\lambda) = 1 + h\lambda + \frac{1}{2}(h\lambda)^2 + \frac{1}{6}(h\lambda)^3 + \frac{1}{24}(h\lambda)^4,$$

i.e., that $w_{i+1} = Q(h\lambda)w_i$ when applied to the test equation $y' = \lambda y$.

Solution:

We simply replace any $f(t, w)$ with λw. In detail,

$$\begin{aligned} k_1 &= h\lambda w_i \\ k_2 &= h\lambda(w_i + k_1/2) = \lambda h(w_i + \frac{1}{2}h\lambda w_i) = (\lambda h + \frac{1}{2}(h\lambda)^2)w_i \\ k_3 &= h\lambda(w_i + \frac{1}{2}k_2) = (h\lambda + \frac{h\lambda}{2}(\lambda h + \frac{1}{2}(h\lambda)^2))w_i = (h\lambda + \frac{(h\lambda)^2}{2} + \frac{1}{4}(h\lambda)^3)w_i \\ k_4 &= \lambda h(w_i + k_3) = (h\lambda + (\lambda h)^2 + \frac{(h\lambda)^3}{2} + \frac{1}{4}(h\lambda)^4)w_i \\ w_{i+1} &= w_i + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \\ &= w_i + \frac{1}{6}(h\lambda w_i + 2((\lambda h + \frac{1}{2}(h\lambda)^2)w_i) + 2((h\lambda + \frac{(h\lambda)^2}{2} + \frac{1}{4}(h\lambda)^3)w_i) + (h\lambda + (\lambda h)^2 + \frac{(h\lambda)^3}{2} + \frac{1}{4}(h\lambda \\ &= \left(1 + \lambda h + \frac{1}{2}(\lambda h)^2 + \frac{1}{6}(\lambda h)^3 + \frac{1}{24}(\lambda h)^4\right)w_i \end{aligned}$$

b) Consider the ODE $y'' + 4y' + 13y = 13\cos(t/10)$. Find the homogeneous solution to this ODE (this should be pen-and-paper, no programming involved).

Solution:

The homogenous ODE is $y'' + 4y' + 13y = 0$. We guess $y = e^\lambda t$, plug this in, and divide by y (since $y \neq 0$) to get the polynomial equation for y:

$$\lambda^2 + 4\lambda + 13 = 0$$

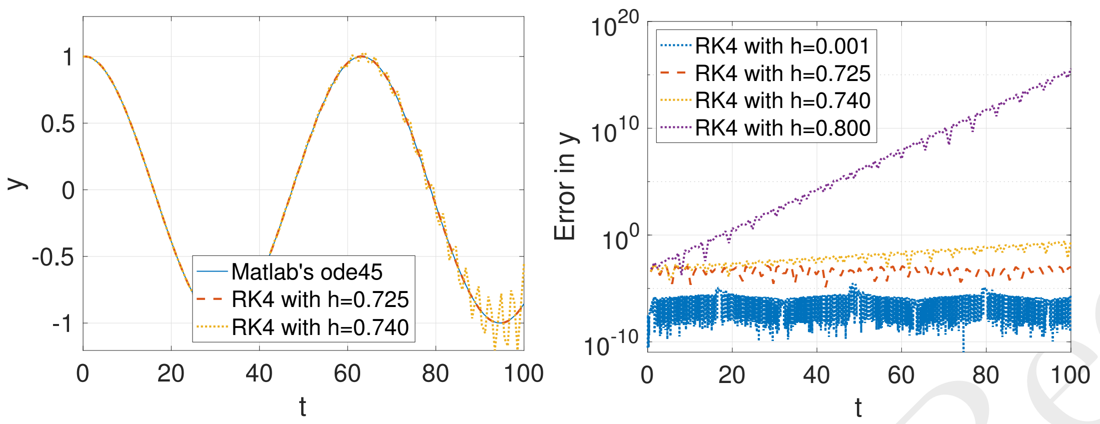

Figure 1: Problem 1c: solving the IVP via RK4 for different stepsizes h (actually numerically solving this was *not* required on the HW). Left: exact solution (approximated using Matlab's solver `ode45`), and RK4 using a stepsize just below and just above the critical stepsize. Right: error, in log-scale, for several values of stepsizes (2 above and 2 below the critical stepsize).

which, via the quadratic formula, has roots $\lambda = -2 \pm 3i$. Hence the homogeneous solution is $\boxed{y(t) = c_1 e^{(-2+3i)t} + c_2 e^{(-2-3i)t}}$ for some constants c_1, c_2. You can also rewrite this as $y(t) = ae^{-2t}\cos(3t+\varphi)$ for some constants a, φ if you like (this might make it more clear that the behavior is a damped sinuisoid), or even as $y(t) = e^{-2t}(a_1\cos(3t) + a_2\sin(3t))$ for constants a_1, a_2.

If this derivation wasn't clear, please review your notes from your ODE course (for APPM 2360, this is chapter 4 in Farlow et al.).

Another way to derive this is to rewrite the ODE as a 2D system of first-order ODEs $\mathbf{y}' = A\mathbf{y}$ where $\mathbf{y} = [y, y']$ and $A = \begin{bmatrix} 0 & 1 \\ -13 & -4 \end{bmatrix}$ and find the eigenvalues of A, which are also exactly $\lambda = -2 \pm 3i$.

c) We're interested in solving the IVP for the above ODE with $y(0) = 1$, $y'(0) = 0$ on the interval $t \in [0, 100]$. Suppose we solve this using RK4 with a stepsize of $h = 3/4$. What can you say about the solution? Explain your answer (again, no programming needs to be involved).

Solution:

There are two parts to this: a relatively straightforward computation (plug λ from part (b) into A from part (a)), and interpretation.

First, the computation. As we've found $Q(h\lambda)$ and we know that for the homogeneous solution we have $\lambda = -2 \pm 3i$, we can plug in h and see if $|Q(h\lambda)|$ is above or below 1. We find $\boxed{|Q(\frac{3}{4} \cdot (-2 \pm 3i))| = 1.1161 > 1}$, so we conclude that this method is *not* absolutely stable for this choice of h and λ. In fact, doing some root-finding, we see that we're absolutely stable iff $h < h_{\text{critical}} = 0.7256$. Note that you should *not* just use $\lambda = -2$; we want the entire complex root. This is a common mistake to make, especially if you rewrote part (b) in terms of sines/cosines.

Now, the interpretation. Our solution is not just $e^{\lambda t}$, but it does include terms like this. So we expect our absolute stabilitiy analysis to be a good predictor of actual behavior. In particular, not being absolutely stable means that the error increases as $T \to \infty$ (though the error goes to 0 as $h \to 0$ since this is a zero-stable method). This part of the question can be graded more leniently. The big issues are not to confuse zero-stability and absolute stability; for this question, the issue is that with a large stepsize, we get spurious components that grow with time and make the numerical solution very inaccurate.

Looking at Figure 1 which shows numerical solutions to the IVP using RK4 (such a plot was not required), we see that's exactly right: the critical stepsize is a great predictor of long-term stability.

Problem 2: Gaussian elimination

a) Implement your own code (in Matlab or Python) to perform Gaussian elimination and then back-substitution to solve a square system of equations $A\mathbf{x} = \mathbf{b}$.[1] Include your code in your homework. For simplicity, you do *not* have to do any kind of pivoting (we'll optimistically assume that we never encounter a zero pivot).

Solution:

See `HW11_GaussianElimination.m` (Matlab) and `HW11_GaussianElimination.py` (Python) at the end of this PDF for code.

b) Let $n = 10$ and define the $n \times n$ matrix $A = [A_{ij}]$ as

$$A_{ij} = \cos\left(\frac{(i-1)(j-\frac{1}{2})\pi}{n}\right) \quad \forall i, j = 1, \ldots, n$$

so it should look like

```
1.0000    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000
0.9877    0.8910    0.7071    0.4540    0.1564   -0.1564   -0.4540   -0.7071   -0.8910   -0.9877
0.9511    0.5878    0.0000   -0.5878   -0.9511   -0.9511   -0.5878   -0.0000    0.5878    0.9511
0.8910    0.1564   -0.7071   -0.9877   -0.4540    0.4540    0.9877    0.7071   -0.1564   -0.8910
0.8090   -0.3090   -1.0000   -0.3090    0.8090    0.8090   -0.3090   -1.0000   -0.3090    0.8090
0.7071   -0.7071   -0.7071    0.7071    0.7071   -0.7071   -0.7071    0.7071    0.7071   -0.7071
0.5878   -0.9511   -0.0000    0.9511   -0.5878   -0.5878    0.9511    0.0000   -0.9511    0.5878
0.4540   -0.9877    0.7071    0.1564   -0.8910    0.8910   -0.1564   -0.7071    0.9877   -0.4540
0.3090   -0.8090    1.0000   -0.8090    0.3090    0.3090   -0.8090    1.0000   -0.8090    0.3090
0.1564   -0.4540    0.7071   -0.8910    0.9877   -0.9877    0.8910   -0.7071    0.4540   -0.1564
```

This is a version of the Discrete Cosine Transform (the DCT-II). Let $\mathbf{x} = (1, 2, \ldots, n)^T$, and define $\mathbf{b} = A\mathbf{x}$. We know $\mathbf{x}$ and we'll keep this in mind for checking the error, but now pretend we don't know $\mathbf{x}$ and we'll solve for $\mathbf{x}$.

Apply your Gaussian elimination code to solve $\mathbf{b} = A\mathbf{x}$ for $\mathbf{x}$, and report the augmented matrix (with 2 decimal places) after it has become upper triangular, and also the entrywise error in your final solution for $\mathbf{x}$.

Solution:

The augmented matrix to 2 decimal places is:

```
1.00     1.00     1.00     1.00     1.00     1.00      1.00      1.00      1.00      1.00  |     55.00
0.00    -0.10    -0.28    -0.53    -0.83    -1.14     -1.44     -1.69     -1.88     -1.98  |    -74.50
0.00     0.00     0.10     0.47     1.22     2.40      3.88      5.42      6.70      7.42  |    227.63
0.00     0.00     0.00    -0.24    -1.34    -4.14     -9.01    -15.32    -21.40    -25.16  |   -662.31
0.00     0.00     0.00     0.00     0.80     5.05     16.35     35.58     57.57     72.54  |   1676.96
0.00     0.00     0.00     0.00     0.00    -3.16    -19.97    -61.45   -120.60   -165.99  |  -3395.63
0.00     0.00    -0.00     0.00     0.00     0.00     11.88     67.68    177.18    275.96  |   4978.80
0.00     0.00     0.00     0.00     0.00     0.00      0.00    -34.26   -154.86   -294.56  |  -4613.41
0.00     0.00    -0.00     0.00     0.00     0.00      0.00      0.00     56.96    165.30  |   2165.58
0.00     0.00     0.00     0.00     0.00     0.00      0.00      0.00      0.00    -31.96  |   -319.62
```

The setup (in Matlab) looks like

```matlab
rng(1); n = 10; A   = zeros(n);
for row = 1:n
    A(row,:)   = cos( pi*((0:n-1)+1/2)*(row-1)/n );
```

[1]For Python users, be aware that `numpy` distinguishes between a pure vector of length n (which has "shape" `(n,)`) and a row vector of length n (shape `(1,n)`) and a column vector of length n (shape `(n,1)`). Often a pure vector is treated like a column vector, but not always. You can convert between the different shapes using things like `numpy.reshape`, `numpy.atleast_2d`, and `.T` (to transpose things). Other helpful `numpy` things may include `numpy.hstack` or `numpy.concatenate`, and `@` for matrix multiplication.

```
end
x   = (1:n)';
b   = A*x;
```

The error is (if students took a norm, that's OK too; the Euclidean norm should be about `9.4907e-12`):

```
  1.0e-11 *

  -0.3268
   0.6281
  -0.3875
   0.0542
   0.0508
   0.0148
  -0.1429
   0.2934
  -0.3370
   0.1529
```

What you should *not* do is only print out your $\mathbf{x}$ to a few decimal places and conclude that it's exactly correct (and be aware that usual Matlab/python print functions might round things for you). If it's looking like you have the exactly right answer, look at the difference between your answer and the true $\mathbf{x}$ — that's exactly what we wanted you to display ("entrywise error in your final solution for $\mathbf{x}$").

c) Let $n = 1000$ and A be a $n \times n$ matrix with entries from the standard normal distribution[2], and again define $\mathbf{x} = (1, 2, \ldots, n)^T$ and define $\mathbf{b} = A\mathbf{x}$. How long does it take your Gaussian elimination code to solve this system? And what's the error (in terms of Euclidean norm, `norm` in Matlab and `numpy.linalg.norm` in Python)?

Hint: for timings, in Matlab you can use `tic` and `toc`; in Python, you can import the `time` module and use `time.perf_counter()`, or

```
from timeit import time
%time [your command here]
```

which is an IPython line magic (so works in IPython and Jupyter). Fun fact: IPython was created as a side project by CU Boulder student Fernando Pérez in 2001.

Solution:

It will depend on your computer, but something around 2 seconds (probably between 1 and 3 seconds). There's no wrong answer, but anything < 1 second is suspicious, and anything over 10 seconds means you probably could greatly improve your implementation with a little bit of work. But, if it did take (say) a minute, that's OK, you probably used all `for` loops and didn't do any dot products. The point of this wasn't to write optimized code.

The norm of the error is 1.2722×10^{-5} for my implementation, and so it should probably be within a factor of 10 of this for yours.

Note: it is important that A is a random matrix like the one we defined, as then the lack of pivoting gives us a lower accuracy solution but it's not catastrophic. For other choices of A, such as the DCT matrix we used in the 10×10 example, our algorithm is terribly unstable and will give unusable results once the size is large (say, about 50×50) — this is all because we're not pivoting.

d) Solve the same system of equation as in (c) but use Matlab's/Python's default solvers, and report the time. For Matlab, this is "backslash" `A\b` aka `mldivide`, and in Python, this is `scipy.linalg.solve`. Also report the error, and comment on the differences between (d) and (c).

[2] in Matlab, `randn(n);` and in Python, `from numpy.random import default_rng` then `rng = default_rng()` then `A = rng.standard_normal((n,n))`

Solution:

Again, there's no right answer, but probably something like 0.001 to 0.03 seconds.

The norm of the error is 1.4114×10^{-8} for my computer (using Matlab), and so it should probably be within a factor of 10 of this for yours.

We see that the builtin implementations are way faster, and also more accurate (due to pivoting most likely). In particular, the builtin implementation is $10\times$ to $1000\times$ more accurate and $10\times$ to $1000\times$ faster.

Problem 3: LU factorization verse the inverse We're interested in solving the square $n \times n$ system of equations $A\mathbf{x} = \mathbf{b}$ using standard techniques; we'll do some programming, but it will be high-level programming. To recall `LU`-factorization, we write

$$A\mathbf{x} = \mathbf{b}, \quad A = \underbrace{P \cdot L \cdot U}_{\text{pivoted LU factorization}}, \quad \text{so} \quad P \cdot L \cdot \underbrace{U \cdot \mathbf{x}}_{\mathbf{y}} = \mathbf{b}$$

where P is a permutation matrix, either given as an explicit matrix (and it is orthogonal, so $P^T = P^{-1}$ hence $LU\mathbf{x} = P^T\mathbf{b}$) or as a permutation vector `p` of the indices so we can apply $P^T b$ as `b(p)` (Matlab) or `b[p]` (Python). Then we first solve $L\mathbf{y} = P^T b$ using *forward-substitution*, and then solve $U\mathbf{x} = \mathbf{y}$ using *backward-substitution*. Be careful: Matlab and Python have different conventions for returning P — Matlab actually returns P^{-1} while Python returns P.

a) Write high-level code that solves $A\mathbf{x} = \mathbf{b}$ using these three methods:

   i. (Highest level) Use "backslash" `A\b` aka `mldivide` (Matlab) or `scipy.linalg.solve` (Python)

   ii. (LU with pivoting) Perform an LU factorization and then solve via backsubstitution, relying on Matlab or Python libraries for everything. In particular, for Matlab, use either `[L,U,P] = lu(A)` or `[L,U,p] = lu(A,'vector')` (slightly faster), and figure out how to use the permutation matrix; then you can call something like `U\y` and Matlab will know that `U` is upper triangular and use backsubstitution. For Python, use either `scipy.linalg.lu(A,permute_l=False)` followed by some `scipy.linalg.solve_triangular` (for the forward/backward substitution; sometimes with the flag `lower=True`) and applying the inverse permutation to b; or use `scipy.linalg.lu_factor(A)` followed by `scipy.linalg.lu_solve`. Note that neither Matlab nor `scipy` has an LU factorization function that doesn't pivot; if it doesn't appear to pivot, it means that it really has pivoted but just incorporated the permutation into L.

   iii. (Inverse) Solve via an explicit inverse of A, using `inv` (Matlab) or `scipy.linalg.inv` (Python).

Show your code (each method might only be one or two lines of code). It's a good idea to check your code on a small problem with a known answer to make sure all these methods work.

Solution:

Matlab: (i) is just `A\b` and (iii) is just `inv(A)*b`. For (ii), the code is (preferred)

```
[L,U,p]     = lu( A, 'vector' ); % A(p,:) = L*U, so permute b via b(p)
x  = U\(L\b(p));
```

or

```
[L,U]     = lu( A, 'vector' ); % P*A = L*U, so permute b via P*b
x  = U\(L\(P*b));
```

It is *not* correct to do

```
[L,U]     = lu( A, 'vector' ); % A = L*U
x  = U\(L\b);
```

because while this appears not to do pivoting, it actually absorbs P into L so that it's not lower triangular, so now `L\b` is not done with forward-substitution and is thus slow. So this is accurate but slow.

In Python, (i) is just `scipy.linalg.solve(A,b)` and (iii) is just `scipy.linalg.inv(A)@b`. For (ii), the code is

```
P,L,U = lu(A,permute_l=False)
x  = solve_triangular(U,solve_triangular(L,P.T@b,lower=True))
```

or you can have `scipy` do almost everything for you if you do it this way:

```
LU, p = lu_factor(A)
x = lu_solve((LU, p), b)
```

b) Record the timing and error for each of these methods for a range of sizes n between 1000 and 5000, and plot both the error and the timing; is the timing $O(n^2)$ or $O(n^3)$ or $O(n^4)$? For a given n, generate A, $\mathbf{x}$ and $\mathbf{b}$ as we did in Problem 2c/d. Comment on your findings.

Hint: for timings, in Matlab you can use `tic` and `toc`; in Python, you can import the `time` module and use `time.perf_counter()`.

Solution:

See Fig. 2 for the plots. We find that all methods are about $O(n^3)$ (certainly not $O(n^2)$ or $O(n^4)$; the actual agreement with $O(n^3)$ isn't perfect as this is affected by memory communication, parallelization and such). If we ran to larger n, we'd expect to see closer to $O(n^3)$ (unless the library switches to Strassen's algorithm, which is about $O(n^2.8)$). So correct student answers for this problem include $\boxed{O(n^3)}$ or $\boxed{\text{a bit less than } O(n^3)}$. It's definitely slower than $O(n^2)$.

The error for methods (i) and (ii) should be about the same or exactly the same (in Matlab, they are exactly the same, except for special cases, such as when Matlab guesses that A is symmetric, and then `A\b` switches to a different method like a `LDL` factorization; in Python, you have to explicitly tell `scipy.linalg.solve` when A is symmetric).

The error for the explicit inverse, (iii), should be quite a $\boxed{\text{slightly worse}}$. Furthermore, (iii) should take quite a $\boxed{\text{slightly longer to compute}}$, like twice as long. For student answers, quantitative answers are OK, but we're mainly looking for a general observation, so qualitative answers are also acceptable.

Problem 2a complete code

Here is the complete code.

Matlab

`HW11_GaussianElimination.m`:

```
function x = HW11_GaussianElimination(A,b,displayAugmented)
% x = HW11_GaussianElimination(A,b) implements
%   Gaussian Elimination and then back-substitution to solve the square
%   system of equations A*x=b
% This does NOT do pivoting of any sort, so it can fail
if nargin < 3 || isempty(displayAugmented), displayAugmented = false; end
[m,n]   = size(A);
```

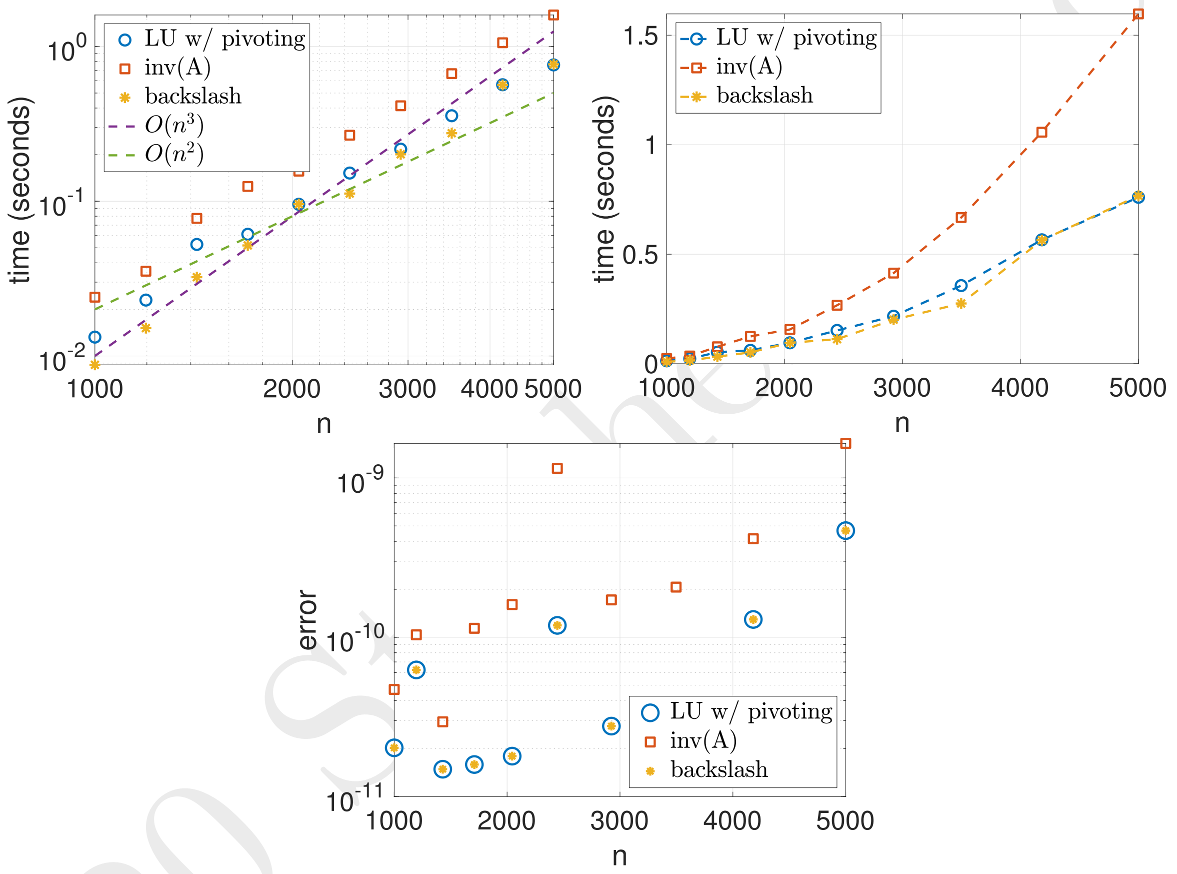

Figure 2: Problem 3b. Top left: timing, log scale, where we can see $O(n^3)$ behavior. Top right: timing, linear scale, so we can easily see that $A^{-1}b$ is about twice as slow. [Students didn't need to have two plots here; they should have at least one plot log-log scale so we can see the $O(n^\alpha)$ behavior]. Bottom: Error. For Matlab, the backslash and LU with pivoting have exactly the same error because the backslash uses LU with pivoting. The error for $A^{-1}b$ can be up to $10\times$ worse than via LU with pivoting.

```
if m ~= n, error('A should be a square matrix!'); end

Ab  = [A,b];    % Augmented matrix
% Gaussian Elimination
for col = 1:n
    row0    = col;
    for row = row0+1:n
        Ab(row,:) = Ab(row,:) - Ab(row,col)/Ab(row0,col)*Ab(row0,:);
    end
end
% For debugging, check that Ab is upper triangular
if displayAugmented
    displayAugmentedMatrix(Ab);
end

% Now, solve via back-substitution
x  = zeros(n,1);
bb  = Ab(:,n+1);
for row = n:-1:1
    x(row) = (bb(row) -  Ab(row,row+1:n)*x(row+1:end) )/Ab(row,row);
end

function displayAugmentedMatrix(Ab)
for row = 1:size(Ab,1)
    for col = 1:size(Ab,2)-1
        fprintf('%8.2f  ',Ab(row,col));
    end
    fprintf('|  %9.2f\n', Ab(row,size(Ab,2)));
end
```

Python

HW11_GaussianElimination.py:

```
import numpy as np
import scipy.linalg

n = 10
A = np.zeros((n,n))
for row in range(n):
  A[row] = np.cos( np.pi*row/n*(np.arange(n)+1/2) )
x = np.arange(1,n+1)
b = A@x

def GaussianElimination(A,b,displayAugmented=False):
  (m,n) = A.shape
  if m != n:
    raise ValueError('A must be a square matrix')
  bShape = b.shape
  b = np.atleast_2d(b).T # if 1D vector, convert to column vector
  # or, b = reshape(b,(10,1))
  Ab = np.hstack((A,b)) # or np.concatenate( ..., axis=1)
  # Gaussian Elimination
  for col in range(n):
    row0 = col
    for row in range(row0+1,n):
      Ab[row,:] -= Ab[row,col]/Ab[row0,col]*Ab[row0,:]
  if displayAugmented:
    matprint( Ab )
  # and solve via back-substitution
  x = np.zeros((n,1))  # np.zeros(n) will cause issues
  bb= Ab[:,-1] # last column of augmented matrix
  for row in np.flip(range(n)): # go backward
    x[row] = ( bb[row] - Ab[row,row+1:n]@x[row+1:] )/Ab[row,row]

  # and put x back in the format that b was in
  return np.reshape(x,bShape)
```

where we used this helper function:

```
def matprint(mat, fmt="g",roundToDecimal=3):
  # from https://gist.github.com/braingineer/d801735dac07ff3ac4d746e1f218ab75
  # Modified to round
  if roundToDecimal is not None:
    mat = np.round(mat,decimals=roundToDecimal)
  col_maxes = [max([len(("{:"+fmt+"}").format(x)) for x in col]) for col in mat.T]
  for x in mat:
    for i, y in enumerate(x):
      print(("{:"+str(col_maxes[i])+fmt+"}").format(y), end="  ")
    print("")
```